AFRL-RI-RS-TR-2008-95
Final Technical Report
March 2008

HIGH PRODUCTIVITY COMPUTING SYSTEMS (HPCS) LIBRARY STUDY EFFORT

University of Tennessee

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. AD60

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2008-95 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
CHRISTOPHER J. FLYNN Work Unit Manager	JAMES A. COLLINS, Deputy Chief Advanced Computing Division Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* MAR 2008	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* Aug 06 – Aug 07

4. TITLE AND SUBTITLE HIGH PRODUCTIVITY COMPUTING SYSTEMS (HPCS) LIBRARY STUDY EFFORT	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER FA8750-06-1-0239
	5c. PROGRAM ELEMENT NUMBER 62303E
6. AUTHOR(S) Jack Dongarra, James Demmel, Piotr Luszczek, Parry Husbands,	**5d. PROJECT NUMBER** AD60
	5e. TASK NUMBER TE
	5f. WORK UNIT NUMBER NN
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Tennessee 1534 White Ave Knoxville TX 37996-1529	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/RITB 525 Brooks Rd Rome NY 13441-4505 Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington VA 22203-1714	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2008-95

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# WPAFB 08-1134

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The research team explores a rich feature set, large algorithmic variety, and detailed implementation considerations for one of the most fundamental computational kernels of computational science: LU factorization of a dense matrix by Gaussian elimination with partial pivoting. For the target implementation platforms and systems, they analyze and compare established shared and distributed memory environments as well as relatively new Partitioned Global Address Space programming languages, which include those coming from the High Productivity Computing Systems (HPCS) project. To give quantitative measures of each hardware platform metrics, combined with implementation characteristics, they compare scalability, raw and relative performance as well as the source code features, functionality, and absolute size breakdown as measured by Source Lines of Code (SLOC).

15. SUBJECT TERMS
LU factorization, scalability, productivity

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Christopher J. Flynn
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	21	**19b. TELEPHONE NUMBER (*Include area code*)** N/A

Table of Contents

List of Tables

1. Overview

In this report we present our research into the implementation of numerical libraries using the proposed HPCS languages. Faced with the fact that the community has very little application experience (the implementations are not yet mature) with these languages, we chose a somewhat atypical approach: perform a case study of parallel Lower, Upper (LU) factorization and determine how this kernel can be implemented in the languages. As such we decided to gather various algorithmic techniques that have been successful and make connections to specific HPCS language features.

We settled on parallel LU factorization for a variety of reasons:

- It is a well known, understandable kernel
- Many implementations exist that span the performance spectrum
- Getting it to perform well in parallel on distributed memory machines reveals many programming issues, solutions to which aren't well represented in traditional languages.

In Section 2 we give a short description of the algorithm and outline some of the roadblocks to high performance. Section 3 presents some of the abstraction issues that arise when comparing the implementation of different versions of the algorithm in different languages. Sections 4 and 5 describe the utilized software and hardware environments, respectively. Section 6 contains our survey of the implementations. In Section 7 we relate development effort to performance. We detail our observations regarding implementing a high performance LU code in an HPCS language in Section 8. Sections 9-11 discuss issues such as portability, automatic performance tuning and semi-automatic program tuning. In Section 12 we describe the types of software development tools needed for the DARPA platforms. Section 13 is reserved for discussing feedback given to HPCS hardware and language developers while Section 14 discusses results from the study workshop. And finally, Section 15 contains the conclusion.

2. LU Factorization and its Implementation Challenges

LU factorization attempts to decompose a general matrix A into a unit lower triangular (L) and upper triangular matrix (U). Row permutations are typically used for numerical stability and so a permutation matrix (P) is also generated such that $LU=PA$. The basic algorithm for this is shown below, assuming a square n x n matrix A:

```
for $i = 1$ to $n-1$
    ● find maximum absolute element in column $i$ below the diagonal
    ● swap the row of maximum element with row $i$
    ● scale column $i$ below diagonal by $1/A(i,i)$
      $L(i,i)=1$
      for $j = i+1$ to $n$
         $L(j,i)=A(j,i)/A(i,i)$
    ● Set row $i$ of $U$
      for $j = i$ to $n$
        $U(i,j)=A(i,j)$
```

- Perform a "trailing matrix update", i.e. update the part of the matrix below and to the right of *A(i,i)*

  ```
  for $j=i+1$ to $n$
    for $k = i+1$ to $n$
      $A(j,k) = A(j,k)-L(j,i)*U(i,k)$
  ```

  This step can equivalently be expressed as a "rank-one update":

  ```
  $A(i+1:n,i+1:n) = A(i+1:n,i+1:n)$ -
                   $L(i+1:n,i)*U(i,i+1:n)$
  ```

In order to achieve high performance through the use of Level 3 Basic Linear Algebra Subprograms (BLAS-3) (matrix-matrix) [10, 11, 12, 13] operations, implementers usually express the algorithm in block form. Challenges to high performance in a parallel setting include management of the following:

- Communication for the row exchanges, updates to L and U, and the trailing matrix updates
- The dependencies in the algorithm

At this point it is interesting to note that sometimes the abstractions provided by a particular environment might inhibit optimization possibilities. A primary example of such inhibition is the set of design decisions that led to the creation of the Scalable Linear Algebra Package (ScaLAPACK) library [2, 8].

The ScaLAPACK library implementers focused on two primary aspects of large scale parallel computing: scalability and portability. The former was addressed by the choice of appropriate parallel data organization and use of established parallel algorithms that could be proven to scale on distributed memory computers. However, the latter aspect reduced the available optimizations to a subset that can be implemented on major variants of parallel hardware. Consequently, the ScaLAPACK code employs a lock-step method that is characterized by heavy synchronization and lack of overlap of communication and computation in the temporal sense (in the spatial sense there exists some overlap as some of the processors are computing while others are communicating data between each other). As a result, ScaLAPACK is easily ported on any existing parallel platform, but its performance can be easily matched and often exceeded by codes targeted at a specific architecture.

3. Mapping to languages & Software Metrics

In this Section we discuss how we developed metrics that guide us through implementations in languages at differing levels of abstraction, the key criticism leveled against using source lines of code (SLOC). In the survey to follow we augment traditional SLOC counts with an indication of the various helper abstractions that were used. These abstractions can either be serial or parallel. In the serial case we primarily have matrix abstractions: use of the familiar "triplet" notation for indexing, built-in matrix operators (\ - backslash, for example, in MATLAB), and "advanced" object oriented features. In addition, we assume that uniprocessor BLAS are provided. The parallel space is more diverse. Languages can provide some subset of any of the following:

- First class distributed arrays
- A global address space
- Data parallelism
- Multithreading
- Atomic transactions
- Advanced synchronization (single/sync variables, clocks, etc.)
- Parallel Matrix Abstractions such as the Parallel BLAS (PBLAS) [2] and Basic Linear Algebra Communication Subprograms (BLACS) [14].

For those implementations that are concerned with high performance, we also measure the best performance attained (absolute and % of peak), the number of processors on which this was measured (an indication of scalability) and, where available, uniprocessor performance (which tells us something about parallel overheads).

4. Survey of programming languages and environments used in the study

The following languages and programming environments were used in this study:

1. MATLAB is a high level language and programming environment with built-in multidimensional arrays. It is a commercially supported product available on a variety of modern processors and operating systems. It is particularly well suited for numerical computations including the LU factorization presented in this report.
2. Octave is a high level language that aims to be an open source implementation of MATLAB. The current version of Octave allows implementation of the LU factorization algorithm presented in this report.
3. Python is a high level language that has an open source implementation. Python combined with an open source numerical extension can be used to implement the LU factorization algorithm presented in this report.
4. CAF stands for Co-Array Fortran. It is an extension of the Fortran language to allow a partitioned global address space paradigm. The extensions defined by CAF are planned to be incorporated in the Fortran 2008 standard. CAF is currently implemented on vector supercomputers such as Cray X1 and clusters of reduced instruction set computer (RISC) processors that use an interconnect fabric with support for one-sided communication.
5. UPC stands for Unified Parallel C [16, 23]. It is an extension of the C programming language to support a partitioned global address space paradigm. Open source implementations of UPC support a wide spectrum of systems including high end supercomputers as well as commodity clusters with commodity interconnects.
6. X10 [15] is a research language developed by IBM for the HPCS program. The language is based on Java syntax with multiple extensions to allow partition global address space programming. Current implementations integrated well with the Eclipse programming environment and generate Java Virtual Machine bytecode as well as native code for performance.
7. Chapel [7] is a research language developed by Cray for the HPCS program. The language is based on the syntax of the Z programming language. There are many constructs to allow partition global address space programming and manipulation of

multidimensional arrays. The current implementation focuses on generating native code for performance.

8. Fortress [1] is a research language developed by Sun Microsystems for the HPCS program. The language has many constructs to allow partition global address space programming as well as other common programming paradigms such as Object Oriented Programming. The current implementation focuses on generating bytecode for the Java Virtual Machine.
9. HPF stands for High Performance Fortran. It is an extension of the Fortran programming language that transforms sequential code into Single Instruction Multiple Data (SIMD) program by using inline comments. As such, the resulting code can run both sequentially with a standard Fortran compiler that ignores the inline comments but it can also run in parallel by taking advantage of the user-supplied parallelization hints that come from the inline comments.
10. Fortran 77 is the language chosen for implementation of widely used numerical libraries including BLAS, LAPACK, and ScaLAPACK. Even though the Fortran 77 standard has been superseded by new versions of the Fortran standard, it is still widely supported and most compilers are able to compile code written in Fortran 77 on most of today's computing platforms.
11. Titanium is a programming language that is a superset of Java. The additional syntax and libraries allow partitioned global address space programming on top of the global address space networking (GASnet) communication substrate. Titanium programs are compiled into native code for maximum performance but still offer many benefits of bytecode and execution inside Java Virtual Machine.
12. C is the implementation language used by the High Performance Linpack code. A reference implementation of a scalable version of the Linpack benchmark [20, 22] requires an implementation MPI library [19, 21] for communicating between processors of a parallel computer. The linker and the runtime of the C programming language is usually used by all other languages, and C is often used as the compilation target for many of the languages used in this study.
13. Cilk is an extension of the C programming language that is targeted for symmetric multiprocessing (SMP) computers and lets the programmer specify parallelism in the code by use of new keywords. The new keywords are meant to give the Cilk runtime hints of parallelism inherent in the code. When the new keywords are removed, Cilk code reduces to standard C. Cilk runtime implements light-weight threading and is one of the first to use the work-stealing technique that has been recently popularized by Intel's Thread Building Blocks.
14. The Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) project aims to address the critical and highly disruptive situation that is facing the Linear Algebra and High Performance Computing community due to the introduction of multi-core architectures [4, 5, 6, 18].

5. Survey of hardware platforms used in the study

The following hardware platforms were used to obtain performance results for this study:

1. Cray X1 is a computer that features vector processors connected by a proprietary interconnect called NUMAlink. The machine is an example of hardware supported global address space. Cray X1 uses the NUMAlink interconnect in a slightly different way than SGI machines do; the cache coherency protocol is the main difference.

2. SGI Altix is a computer that features Intel Itanium processors connected by a proprietary interconnect called NUMAlink. The machine is a ccNUMA (cache coherent Non-Uniform Memory Access) architecture; the main memory access times vary depending on where the memory modules are located and the communication happens via a cache coherency protocol.
3. Itanium cluster with Quadrics interconnect features a high end Intel Itanium processor (a very long instruction word (VLIW) architecture) with a proprietary interconnect optimized for message passing interface (MPI) applications.
4. BlueGene/L is a supercomputer architecture that features a low power PowerPC 440 processor and a proprietary torus interconnect. It is a low power design that can scale to hundreds of thousands of processors.
5. Intel Clovertown is an Intel Core 2 architecture that features 4 cores on a single chip. The tested system had two chips totaling 8 cores. It allows multithreaded programming as well as explicit message passing paradigm.
6. Intel Pentium 4 cluster with an Ethernet interconnect combines a commodity processor with a commodity interconnect. The primary programming model is message passing.

6. Survey of implementations

It is of course arguable how representative such codes are, but the fact that we can easily obtain versions of this algorithm for current and future languages are of interest to HPCS. We present our findings in Table 1 below (the following is a description of the table column headers).

1. **Language:** The main language used for the implementation
2. **Author**: the person who wrote the code
3. **Method**: method used to factorize
    a. Vectorized (calling BLAS-1)
    b. Blocked (calling BLAS-3)
    c. Recursive
    d. Parallel
    e. 1-D, 2-D
    f. Local factorization variants...
    g. Library-based (calling optimized library, perhaps written in a different language)
4. **Pivoting**: is partial pivoting done?
5. **Blocking**: are blocked calls to BLAS made?
6. **Driver**: is driver code included with matrix generation, etc?
7. **SLOC**: number of lines in editor (excluding large blocks of comments)
8. **Distribution**: parallel distribution type (or 0-D for sequential codes)
9. **Lookahead**: Can the code overlap panel factorizations with trailing matrix updates?
10. **Dist**. **Mem?**: Can this code run on distributed memory machines?
11. **Reuse L,U**: Can L and U be reused for further solves after the factorization is complete?

12. **Features**: Any other important features of the code. For example, examples suitable for teaching purposes are marked as “simple”.

Table 1. Findings

Language	**Author**	**Method**	**Pivot-ing**	**Block-ing**	**Driver**	**SLOC**	**Dist**	**Look-ahead**	**Dist. Mem?**	**Reuse L,U**	**Features**
MATLAB	Cleve Moler	Outer product, row-wise	Yes	No	No	37	0-D	No	No	Yes	Simple
Octave	Jason Riedy	Recursive	Yes	Yes	No	130	0-D	No	No	Yes	Algorithm by Sivan Toledo
Python	Piotr Luszczek	Outer product	Yes	No	No	40	0-D	No	No	Yes	Simple
Python	Piotr Luszczek	Outer product	Yes	Yes	No	95	0-D	No	No	Yes	Library
CAF	Robert Numrich	Outer product	Yes	No	Yes	1000	2-D	No	Yes	Yes	Simple, long
CAF	John Reid	Outer product	Yes	Yes	Yes	200	1-D	No	Yes	Yes	Simple
CAF	Robert Numrich	Outer product	Yes	Yes	Yes	120	2-D	No	Yes	Yes	CafLib, SLOC 9222
UPC	Parry Husbands	Outer product	Yes	Yes	Yes	5100	2-D	Yes (Dynamic)	Yes	U, not L	Fast
UPC	Calin Cascaval	Outer product	Yes	Yes	Yes	536	2-D	No	Yes		Simple
X10	Vivek Sarkar	Outer product	Yes	No	Yes	167	2-D	No (?)	Yes*	Yes	Simple
Chapel	Brad Chamberlain	Outer product, row-wise	Yes	No	No	40	0-D	No (?)	Yes*	Yes	Simple
Fortress	Guy Steele, Jan Willem-Massen	Outer-product, row-wise	Yes	No	Yes	100	0-D	No (?)	Yes*	Yes	Simple
HPF	M. Nakanishi	Outer product	Yes	No	No	70	1-D	No (?)	Yes	Yes	Simple
HPF	Anotine Petitet	Outer product	Yes	Yes	Yes	25	2-D	No (?)	Yes	Yes	Library

LINPACK	Cleve Moler	Outer product, vectorized	Yes	No	No	60	0-D	No	No	Yes	dgefa
LAPACK	LAPACK team	Outer product	Yes	Yes	No	100+ 100	0-D	No	No	Yes	Dgetrf dgetf2
ScaLAPACK	Antoine Petitet	Outer product	Yes	Yes	No	180+ 140	2-D	No	Yes	Yes	PDGETRF PDGETF2
HPL	Antoine Petitet	Outer product	Yes	Yes	Yes	5000+	2-D	Yes (Static)	Yes	U, not L	
Titanium	Simon Yau	Outer product	No	Yes	Yes	388		No	Yes		
C	PLASMA team	Outer product	Yes	Yes	Yes	400	2-D	Yes (Dynamic)	No	Yes	Multithreaded
C	Panziera and Baron	Outer product	Yes	Yes	Yes		2-D	Yes (Dynamic)	Yes	U, not L	Multithreaded (up to 512p) + MPI
Cilk	Bradley Kuszmaul	Recursive	Yes	Yes	Yes	266	0-D		No		Multithreaded

* global address space rather than distributed address space

Because the level of abstraction varies widely among the various languages, it is beneficial to comment on the services and abstractions that each language provides.

Table 2. Services & Abstractions of languages

Language	**Services & Abstractions**
Matlab	triplet, BLAS as operators, data parallel abstraction
Python	triplet, BLAS as operators, data parallel abstraction
CAF	triplet, first class distributed arrays, global address space
UPC	first class distributed arrays, global address space
X10	first class distributed arrays, global address space, data parallel + multithreading, “clocks”, atomics, “advanced” OO
Chapel	first class distributed arrays, global address space, data parallel + multithreading, atomics, “advanced” OO
Fortress	first class distributed arrays, global address space, data parallel + multithreading, atomics, “advanced” OO
HPF	triplet, first class distributed arrays, data parallel
f77/f90	triplet, PBLAS, BLACS
Titanium	first class distributed arrays, global address space
Cilk	multithreading,

Table 1 above and Table 3 below, which relates languages used with performance, are discussed in greater detail in section 7.

Table 3. Performance of those codes that strive for high performance.

Language	Author	Best Performance GFlop/sec	p	Machine	% peak	Best 1p %peak
CAF	Robert Numrich	509	60	Cray X1	71.0	92.1
UPC	Parry Husbands	2249	512	Itanium/Quadrics	78.4	91.8
UPC	Calin Cascaval	118	256	BG/L	16.4	52.5
HPL	Antoine Petitet	280600	131072	BG/L	76.4	80.1
C	PLASMA team	48.5	8	Intel Clovertown	57.0	70.3
C	Panziera and Baron	51870	10160	SGI Altix Cluster	85.1	90.1
ScaLAPACK	Antoine Petitet	44	64	Intel Pentium 4 cluster	14.3	47.0

Taking LAPACK's code as an example, Table 4 below provides a breakdown of line counts of various sections of the code:

Table 4. Line counts.

	DGETRF	DGETF2	Total	Percentage
Leading comments	36	36	72	24.4%
Blank comments	50	43	93	31.5%
Other comments	19	13	32	10.8%
Total comments	105	92	197	67%
Declarations	11	11	22	7.5%
Argument checking	14	14	28	9.5%
Real work	30	18	48	16%
Total	160	135	295	

Consequently, the total length can be thought of as anywhere from 48 SLOC (for "real work") up to 295 SLOC. And we ignore the code in the library calls to the Basic Linear Algebra Subprograms (BLAS): Double-precision General Rank 1 (DGER), Double-precision Scale (DSCAL), Double-precision Swap (DSWAP), Double-precision General Matrix-Matrix multiply (DGEMM), Double-precision Triangular Matrix Solve Matrix (DTRSM) as well as LAPACK's auxiliary routines: Double-precision LAPACK Auxiliary Swap (DLASWP) and Integer

LAPACK Auxiliary Environment (ILAENV). Furthermore, this hardly captures the level of effort in the PBLAS or BLACS, which were designed with a lot more generality and complexity in mind than needed for ScaLAPACK's Parallel Double-precision General Triangular Factorization (PDGETRF) subroutine alone. In comparison, the Unified Parallel C (UPC) version [17] sacrifices the generality and builds the complexity from scratch, and so comes in last in the SLOC metric (if SLOC could be considered as a metric).

Table 5. SLOC counts for Cilk, UPC, and PLASMA.

Cilk	
Category	**SLOC**
Scheduler	190
Panel Factorization	10
Trailing Matrix Updates	70
Driver	100
Comments	30

UPC	
Category	**SLOC**
Serial Kernels	82
LU	34
Backsolve	51
Trailing Matrix	22

PLASMA	
Category	**SLOC**
Threading Package	215
Panel Factorization	1002
Update to U	110
Trailing Matrix Update	454
Back Substitution	368

7. Relating program development effort to performance

Analysis of tables 1 and 3 gives us insight into the interaction of the development effort and resulting execution performance. In addition, combining this with information from table 2 shows how the software environment can alleviate the effort and increase the performance. The main conclusion is that, on current and future architectures, a high percentage of peak performance can only be achieved with a non-trivial amount of coding regardless of the programming language involved. This emphasizes the importance of high quality software libraries available across the HPCS platforms.

Another important conclusion is that scalable code does not necessarily achieve a high percentage of peak performance regardless of the amount of lines of code involved, if either lower-level computation and communication primitives are not fully used or the hardware does not expose sufficient amount of parallelism and latency hiding. This point will be further stressed by the HPCS platforms, which will have a large computational and communication potential available through multi-faceted programming languages that promise to offer various concurrency primitives. Delivering highly optimized numerical kernels will have to take advantage of both and will involve non-trivial amounts of coding.

Finally, reducing coding effort without sacrificing execution performance requires the ability to freely compose software modules. By so doing we were able to isolate performance critical portions of many of the codes studied. Basic compositing functionality, such as object-oriented programming and dynamic library linking, is promised to be included in HPCS languages and operating system services.

8. Writing in an HPCS Language

From our survey, we can conclude that while pure data parallel approaches to writing LU factorization can produce compact code, they do not perform particularly well. This leads us to consider alternative approaches. Because all of the HPCS languages include task parallel facilities and bearing in mind that the simple alternative of simulating a single program multiple data (SPMD) code such as High Performance Linpack (HPL) is always available, we consider the issues involved in writing task parallel LU factorization codes.

We restrict our attention here to multithreaded implementations which have enjoyed a resurgence in recent years. Because our results indicate that blocking and look-ahead are required for performance, we also focus on these two aspects. Blocking is primarily provided by the matrix abstraction while support for look-ahead is dependent on the parallel control flow and synchronization primitives in the language.

Multithreaded approaches have some potential advantages on distributed memory machines:

- Better communication latency tolerance
- Look-ahead (algorithmic latency tolerance) is dynamic leading to improved machine utilization

There are, however, some costs:

- User control over the schedule is needed in order to minimize parallel execution time.
- User (or system) control over the amount of buffering required in distributed memory machines.

The scheduling issue is paramount for performance. It essentially comes down to scheduling a directed acyclic graph (DAG) of tasks on each of the processors. These tasks correspond to the major operations of the algorithm, and edges between them represent dependencies that must be satisfied before the task can run. In the dense linear algebra case, the tasks and dependencies are statically determined by the matrix size and block size. In more complex algorithms, the tasks and edges may be dynamically determined by the data.

Ultimately the scheduler (either a global or many local ones) must decide, for each processor/core, the "best" task to run at any given time, knowing which dependencies have already been met and some information (flops, running time) about the task pool. The difficulty lies in the definition of "best". There are many, possibly competing requirements:

- The task must advance the parallel execution of the algorithm. The scheduler's decision should delay other tasks as little as possible. This is also known as the "critical path" issue.
- The sequence of tasks run on any given processor/core should incur as few cache misses as possible (this may compete with the previous requirement). Because of the dominance of BLAS-3 operations in LU factorization, this is less of an issue here.
- The tasks must be chosen so that buffer memory is not exceeded.

The definition and implementation of protocols for interacting with schedulers is, however, still a research topic (and so have been excluded from the HPCS languages). As such, schedulers have traditionally been built in an application specific manner using parallel control flow features (spawns and waits) combined with various data structures, such as scoreboards for keeping track of dependencies. Thread priorities are also another way of influencing the scheduler, but to our knowledge this hasn't been widely used in scientific computing codes. We anticipate the use of similar techniques in X10, Fortress, and Chapel. Features in these languages for task control include single and sync variables (for producer consumer relationships), spawns with locality directives, guarded statements (that fire when a condition is satisfied), and atomic regions. These are the basic tools that will be used for constructing schedulers [3].

9. Portability issues

Achieving portability could be a daunting task on the HPCS platforms even though there are only a projected few of them. At the most fundamental level there exist serious nomenclature disparities between the participating hardware vendors, even though the subject matter remaining seems to be quite unified. This issue was brought to the vendors' attention at the language workshop (see Section 14).

A related issue is the programming languages being developed by the HPCS vendors. It would be very hard to deliver a high quality numerical kernel library in more than one programming language. And the difficulty becomes yet greater when considering any non-trivial scientific code. This issue, again, was raised during the language workshop and with vendors' understanding.

A likely portability layer could emerge at the library level; each of the HPCS platforms should offer compatible software interfaces to high quality numerical kernels such as LU factorization. Multi-language banding can already be done in portable fashion and has ongoing support of various DOE sites.

Finally, we also envision some level of portability at the basic runtime level. This will unify operating system and messaging primitives to successfully build the aforementioned libraries in a portable manner. A cross-platform messaging runtime on HPCS platforms is an on-going effort within HPCS as discussed during the language workshop (see Section 14).

10. Automatic performance tuning of numerical kernels

Automatic performance tuning is a viable approach for achieving high performance when dealing with complex computer architectures, which is especially important for HPCS hardware. The optimal performance search space will certainly grow in size as we move the tuning process to future hardware. The code that we have chosen as the primary case study, LU factorization with pivoting, exposes multiple aspects of automation. At the most basic level, automatic tuning addresses the basic computer architecture artifacts such as register file, cache structure, and memory hierarchy. On even more complex processors, the tuning process will inadvertently need search space pruning techniques with a cautious choice of eliminated parameter subspaces. We envision here a hierarchical approach to tuning whereby at different levels of the hierarchy the tuning space is pruned differently, yielding varying amounts of tuning time and possibly different results in performance achieved by the resulting code. We also recognize higher level parameters that will be amenable to automatic performance tuning especially on a brand new computer architecture.

In the case of LU factorization, there can be many algorithmic choices with respect to computational and communication aspects of the kernel. The computational aspect of LU factorization includes the algorithm formulation (inner versus outer product, left- versus right-looking etc.), blocking strategy (1D or 2D, uniform or adaptive, etc.), the dependency graph traversal, and threading and multiprocessing strategies. The communication aspect of LU factorization mostly involves messaging patterns at various stages of the algorithm together with size-tuning and asynchronicity of these patterns [9].

Finally, we recognize the contextual aspect of auto-tuning. Context here means both the dependence of data being operated on as well as interaction with other components of the software and hardware ecosystem. We believe that these issues will have to be addressed at some point as the HPCS hardware and software ecosystem matures. The detailed analysis of this aspect exceeds the scope of this study.

11. Program analysis and semi-automatic program tuning

In addition to fully automated performance tuning, we found during the study that an optimization technique known as guided tuning will be equally important on HPCS systems, as it is already on current high end supercomputer architectures. A part of this process is decomposition of functionality and unrelated performance portions of the code. As mentioned earlier, we find it important to efficiently (in terms of programming effort and the runtime overhead) compose these portions of the code into a final working executable. In addition, computational and communication primitives should also allow this mix and match approach as the developer finds the bottlenecks in the code and attempts to remove them by not only algorithmic changes but also by mixing-in additional software and hardware capabilities of the underlying computing platform. We found it indispensable to be able to gather performance data of a running program and go back to address the performance issues in the appropriate portions of the code. This was both true for our own codes as well as for other developers that lent us their codes for study. And we envision this guided tuning to be important at the initial stages of HPCS platform development when the process of guided tuning will be mostly manual. And later on it will be important for development of semi-automatic software tools such as compilers and program analyses tools.

12. Software environment for future DARPA computing platforms

As should be evident from the analysis above, a working compiler is hardly a sufficient requirement for anything but the most trivial programming task. As the HPCS systems aspire to address the issues of programmability and scale, we have identified three aspects of code development that will need the attention of a productive ecosystem:

1. Code reuse,
2. Executable generation, and
3. Performance analysis.

The first aspect, code reuse, stresses existence and availability of high performance and high numerical quality computational kernels. Even though we have chosen a numerical linear algebra kernel as the main example of this study, we do not envision the users of the future HPCS systems writing this kind of functionality code themselves. Instead, we see the need for such code to be developed for the users so they can focus on software that will more directly address the goals of the HPCS program in terms of scientific progress and national security. Numerical linear algebra kernels for both dense and sparse problems should be developed as soon as prototype hardware becomes available and evolve to successfully reach acceptable performance as the HPCS hardware attains production level status. We believe in open source dissemination of such software, as it will serve three important purposes: as a warrant of portability (if not across few HPCS architectures then at the very least across upgrades of a single architecture), as a teaching tool for the users at high-end performance regimes, and as a starting point for the vendors in developing highly-tuned versions of binaries for one particular generation of hardware architecture.

The second aspect, executable generation, stresses the transparency of code generation as well as both static and dynamic linking. It is important for both low-level kernel and application code developers to understand and influence the compilation process, especially on the HPCS platforms, which will feature unprecedented levels of hardware complexity not excluding multi-tier parallelism and heterogeneous hardware components.

Finally, the third aspect, performance analysis, draws attention to the ability of examining the execution of code both at runtime and post-mortem. We envision here access to hardware performance counters and a software profiling interface at various levels of granularity. As was already stressed before, the current high-end architectures and their interfaces for code execution introspection are only the starting point for the development of performance analysis tools as the HPCS hardware is expected to exceed scale and complexity of currently available solutions.

13. Collaboration with the HPCS hardware and language developers

This study increased our collaboration with the HPCS hardware and language developers. Our primary case study, LU factorization, is part of an important benchmarking effort within HPCS called the High Performance Computing Challenge (HPC Challenge or HPCC). HPCS hardware and software developers participated in this effort, which resulted in exchange of information with respect to hardware potential and software requirements for efficient implementation of the LU kernel.

Despite its superficial simplicity, the LU kernel presents a few numerical challenges that stress compliance with the IEEE 754 floating-point standard. A case in point is the treatment of denormal pivots that, if treated properly, might avoid generation of special floating-point values such as infinities or not-a-number (NaNs - which are undefined results of a floating-point operation).

Finally, our study reiterated with the vendors the need for unifying the nomenclature so that various HPCS systems can be compared with respect to raw performance specifications as well as the achieved performance levels.

14. High performance languages workshop

In January 2007 we participated in a 3-day programming workshop at Rice University. Present at the meeting were representatives from the HPCS vendors, mostly programming language developers. HPCS representatives from academia and government labs were also present.

One of the important topics discussed at the workshop was the common messaging runtime. Such runtimes expose the high-bandwidth and the low-latency of the HPCS platforms to both MPI library implementations as well as the HPCS languages. The effort regarding the runtime is still ongoing.

Another broad topic of the meeting was the language features found in different HPCS languages. The features discussed were:

- multidimensional array syntax and semantics and their relation to other language features
- object orientation including templating, inheritance, and their relation to the built-in type system
- support for IEEE 754 floating-point standard: conformance, implementation and performance.

The organizational items included the discussion about unification of hardware and programming language nomenclatures as well as the time frame for deliverables and the milestone schedule. To summarize the findings, it was recognized that, despite the different wording, all HPCS vendors recognize conceptually similar components in their hardware. The programming languages relate to these components, again using different nomenclature but at the same time very similar at the conceptual level. The development of the HPCS languages was on track with prototype systems already available for the existing hardware platforms.

15. Conclusions

Even with its perceived simplicity, parallel LU factorization presents unique challenges to language designers and library writers. We have shown that scaling up the available hardware resources has to be accompanied by programming language tools. If the tools are not provided, then first, the scaling of the code quickly deteriorates and second, the fraction of the peak performance observed in a sequential environment can never be achieved in a parallel setup. But performance is only one part of HPCS' productivity goal. The other important part is programmer effort in delivering a well performing code. Both the programming language features and a rich set of third party libraries are required to achieve this goal.

16. Acknowledgment

This material is based on research sponsored by DARPA under agreement number FA8750-06-1-0239. The US Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation therein.

17. References

[1] Allen E., Chase D., Hallett J., Luchangco V., Maessen J-W., Ryu S., Steele G. L., and Tobin-Hochstadt S. *The Fortress Language Specification*. Available at http://research.sun.com/projects/plrg/Publications/index.html, 2007

[2] Blackford S., Choi J., Cleary A., D'Azevedo E. F., Demmel J. W., Dhillon I. S, Dongarra J., Hammarling S., Henry G., Petitet A., Stanley K., Walker D. W., and Whaley R. C. *ScaLAPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia, 1997.

[3] Blumofe R. and Leiserson C. "Space-Efficient Scheduling of Multithreaded Computations," *SIAM J. on Computing, 27,* 1 (1998), 202-229.

[4] A. Buttari, J. Dongarra, P. Husbands, J. Kurzak and K. Yelick. "Multithreading for Synchronization Tolerance in Matrix Factorization," To Appear in *Proceedings of the 2007 SciDAC Conference*, Boston, MA, July 2007.

[5] Buttari A., Dongarra J., Kurzak J., Langou J., Luszczek P., and Tomov S. "The Impact of Multicore on Math Software," In *Proceedings of PARA 2006*, Umeå, Sweden, June 2006.

[6] Buttari A., Langou J., Kurzak J., and Dongarra J. *Parallel Tiled QR Factorization for Multicore Architectures*. Technical Report UT-CS-07-598, University of Tennessee, Computer Science Department, July 2007. Also published as LAPACK Working Note 190.

[7] Callahan D., Chamberlain B. L., and Zima, H. P. "The Cascade High Productivity Language," In *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004),* pages 52-60. IEEE Computer Society, 2004.

[8] Choi J., Dongarra J., Ostrouchov S., Petitet A., Walker D., and Whaley, R.C. "The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines," *Scientific Programming, 5,* (1996), 173-184.

[9] Cicotti P. and Baden S. "Asynchronous programming with Tarragon," In *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing*, June 19-23 2006.

[10] Dongarra J., Du Croz J., Duff I., and Hammarling S. *Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms*. ACM Transactions on Mathematical Software, 16:1-17, March 1990.

[11] Dongarra J., Du Croz J., Duff I., and Hammarling S. *A set of Level 3 Basic Linear Algebra Subprograms.* ACM Transactions on Mathematical Software, 16:18-28, March 1990.

[12] Dongarra J, Du Croz J., Hammarling S., and Hanson R. *Algorithm 656: An extended set of FORTRAN Basic Linear Algebra Subprograms.* Transactions on Mathematical Software, 14:18-32, March 1988.

[13] Dongarra J, Du Croz J., Hammarling S., and Hanson R. *An extended set of FORTRAN Basic Linear Algebra Subprograms.* ACM Transactions on Mathematical Software, 14:1-17, March 1988.

[14] Dongarra J. and Whaley R. C. *A user's guide to the BLACS v1.1.* Technical Report UT-CS-95-281, University of Tennessee Knoxville, March 1995. LAPACK Working Note 94 updated May 5, 1997 (VERSION 1.1).

[15] Ebcioglu K., Saraswat V., and Sarkar, V. "X10: an Experimental Language for High Productivity Programming of Scalable Systems," In *Proceedings of the P-PHEC 2005 Workshop*, held in conjunction with HPCA 2005, 2005.

[16] El-Ghazawi T., Carlson W., Sterling T., and Yelick K. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2005.

[17] Husbands P. and Yelick K. "Multi-Threading and One-Sided Communication in Parallel LU Factorization," To Appear in *Proceedings of SC 07*, November 2007

[18] Kurzak J. and Dongarra J. *Implementing Linear Algebra Routines on Multi-Core Processors with Pipelining and a Look Ahead.* Technical Report UT-CS-06-581, University of Tennessee, Computer Science Department, 2006. Also published as LAPACK Working Note 178.

[19] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* The International Journal of Supercomputer Applications and High Performance Computing, 8, 1994.

[20] Panziera J.-P. and Baron J. "A Highly Efficient Linpack Implementation Based on Shared-Memory Parallelism," In *Proceedings of the 2005 International Supercomputer Conference*, 2005.

[21] Snir M., Otto S., Huss-Lederman S., Walker D., and Dongarra J. *MPI: The Complete Reference - 2nd Edition: Volume 1*. The MIT Press. ISBN 0-262-57123-4, 1998.

[22] The Top 500 Supercomputer Sites. Available at: http://www.top500.org, 2007.

[23] UPC Consortium. UPC Language Specification, v1.2. Available at: http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf, 2005.

Acronyms

Explanation of acronyms used in this report:

• BLAS – Basic Linear Algebra Subprograms
• BLAS-1 – Level 1 BLAS: operations on vectors
• BLAS-3 – Level 3 BLAS: matrix-matrix operations
• BLACS - Basic Linear Algebra Communication Subprograms
• CAF – Co-array Fortran
• DAG - Directed Acyclic Graph
• DARPA – The Defense Advanced Research Projects Agency
• DGEMM – Double-precision General Matrix-Matrix multiply
• DGER – Double-precision General Rank 1 Update (BLAS)
• DLASWP – Double-precision LAPACK Auxiliary Swap
• DSCAL – Double-precision Scale (BLAS)
• DSWAP - Double-precision Swap (BLAS)
• DTRSM – Double-precision Triangular Matrix Solve Matrix (BLAS)
• GASnet – Global Address Space Networking
• HPCC – High Performance Computing Challenge
• HPCS – High Productivity Computing Systems
• HPF – High Performance Fortran
• HPL – High Performance Linpack benchmark
• ILAENV – Integer LAPACK Auxiliary Environment
• LAPACK – Linear Algebra PACKage
• Linpack – LINear PACKage: a set of Fortran subroutines for numerical linear algebra; also a benchmark based on one of the Linpack subroutines
• LU – Lower Upper
• MPI – Message Passing Interface
• PBLAS – Parallel BLAS
• PDGETRF – Parallel Double-precision General Triangular Factorization (ScaLAPACK)
• PLASMA – Parallel Linear Algebra for Scalable Multi-core Architectures
• RISC – Reduced Instruction Set Computer
• ScaLAPACK – Scalable LAPACK
• SIMD - Single Instruction Multiple Data
• SLOC – Source Line of Code
• SMP – Symmetric Multiprocessing
• SPMD – Single Program Multiple Data
• UPC – Unified Parallel C
• VLIW – Very Long Instruction Word